Table of Contents

Using Parts of the EDM from within ROOT

WARNING: this twiki is outdated (particularly the use of unnamed macros in fwlite is seriously discouraged, the correct twiki is: WorkBookFWLiteExamples

Complete: 

Goal of this page

This page describes methods to access parts of the EDM from within ROOT.

Enabling FWLite

To enable FWLite, issue the following commands to ROOT

In CMSSW_7_4_X and prior releases:

```
gSystem->Load("libFWCoreFWLite")
FWLiteEnabler::enable();     // AutoLibraryLoader::enable() in CMSSW_7_4_X and prior releases
```

In CMSSW_7_5_X and later releases:

```
gSystem->Load("libFWCoreFWLite")
FWLiteEnabler::enable()
```

In releases using ROOT5 and prior releases of ROOT, the `enable()` call is needed so that libraries will be automatically loaded as needed.

Once the dictionary has been loaded, the member functions of the class can be used from TBrowser, TTree::Draw, the ROOT command line or from within a Macro.

In releases using ROOT6, libraries will be loaded as needed even without the `enable()` call. Nevertheless, the enable() call is still required for edm::Ref* objects to work properly with FWLite.

Using an edm::Ref, edm::RefProd, etc. from ROOT

The edm::Ref* family of classes are used by the EDM to link data in separate areas (i.e. ROOT branches) of the edm::Event. For example, an reco::Track contains an edm:Ref which refers to the appropriate edm::TrackExtra instance which is associated with that particular reco::Track.

Using the `AutoLibaryLoader::enable()` or `FWLiteEnabler::enable()` makes edm::Ref* objects work from within ROOT.

Structure of a Working Macro

When building a macro, there is a particular order of calls to ROOT methods which appears to be required in order for ROOT to function properly

Using a TFile

Using fwlite::Event (New in CMSSW_1_5_0)

The fwlite::Event allows you to use the same information you use when accessing data in cmsRun using edm::Event.

1. Start the enabler or autoloader
2. create the TFile
3. load in the helper library `gSystem->Load("libDataFormatsFWLite")`
4. include "DataFormats/FWLite/interface/Handle.h" to get the `fwlite::Handle`
5. create an `fwlite::Event` by passing to the constructor a pointer to the TFile
6. create a for loop
    1. start loop by calling `toBegin()` on the `fwlite::Event`
    2. for each iteration of the loop call `atEnd` on the `fwlite::Event`
    3. at the end of each iteration, increment the `fwlite::Event` by using the operator++ method
7. when looping over the events
    1. create an `fwlite::Handle< ... >` where the template argument is the C++ class you want to get from the event
    2. call the `getByLabel` method of the `fwlite::Handle` passing it the event and the strings used to denote the object

An example macro template is shown below

```
{
   gSystem->Load("libFWCoreFWLite.so");
   FWLiteEnabler::enable();     // AutoLibraryLoader::enable() in CMSSW_7_4_X and prior releases
   gSystem->Load("libDataFormatsFWLite.so");

   #include "DataFormats/FWLite/interface/Handle.h"
   TFile file("....root");

   fwlite::Event ev(&file);

   for( ev.toBegin();
          ! ev.atEnd();
          ++ev) {
       fwlite::Handle<std::vector<...> > objs;
       objs.getByLabel(ev,"....");
      //now can access data
      std::cout <<" size "<<objs.ptr()->size()<<std::endl;
      ...
   }
}
```

Using fwlite::ChainEvent

Using fwlite::ChainEvent is very trivial and comparable to using fwlite::Event:

```
{
   gSystem->Load("libFWCoreFWLite.so");
   FWLiteEnabler::enable();     // AutoLibraryLoader::enable() in CMSSW_7_4_X and prior releases
   gSystem->Load("libDataFormatsFWLite.so");

   #include "DataFormats/FWLite/interface/Handle.h"
   std::vector<std::string> files;
   files.push_back("file1.root");
   files.push_back("file2.root");

   fwlite::ChainEvent ev(files);

   for( ev.toBegin();
          ! ev.atEnd();
```

```
          ++ev) {
       fwlite::Handle<std::vector<...> > objs;
       objs.getByLabel(ev,"....");
      //now can access data
      std::cout <<" size "<<objs.ptr()->size()<<std::endl;
      ...
   }
}
```

Compling the Macro with ACliC

There are several pieces of information you need to know about how ROOT compiles a macro in order to be successful.

1. ROOT runs the CINT interpreter over the macro before compiling it in order to determine if it needs to generate dictionaries for some of the classes/functions mentioned in the macro. Unfortunately many of CMS's headers contain code that CINT can not parse. This means we must hide those headers from CINT but make sure they are visible to the compiler.
2. After compiling the code, ROOT links the compiled Macro against all libraries which have been loaded. If the macro uses a function or variable from a library which has not yet been loaded, then ROOT will issue a 'missing symbol' error.

With the above in mind, here are the steps needed

1. Protect all headers using a

   ```
   #if !defined(__CINT__) && !defined(__MAKECINT__) #endif
   ```

   block
2. Make the macro block into a function with the same name as the file (this keeps CINT from trying to fully parse the internals of the routine)
3. In ROOT
   1. load and start the enabler ot autoloader
   2. load `libDataFormatsFWLite`
   3. create a `TFile` from one of the files you want to read. This will cause all the libraries for every class in the file to be loaded.
   4. compile/link/execute the macro by doing `.x <filename>++`

An example macro template is shown below. The example assume the file is named print_data.C

```
#if !defined(__CINT__) && !defined(__MAKECINT__)
#include "DataFormats/FWLite/interface/Handle.h"
#include "DataFormats/FWLite/interface/Event.h"
//Headers for the data items
...
#endif
void print_data() {
  TFile file("....root");

   fwlite::Event ev(&file);

   for( ev.toBegin();
          ! ev.atEnd();
          ++ev) {
       fwlite::Handle<std::vector<...> > objs;
       objs.getByLabel(ev,"....");
      //now can access data
      std::cout <<" size "<<objs.ptr()->size()<<std::endl;
      ...
   }
```

```
}
```

Then in ROOT one does

```
gSystem->Load("libFWCoreFWLite.so");
FWLiteEnabler::enable();     // AutoLibraryLoader::enable() in CMSSW_7_4_X and prior releases
gSystem->Load("libDataFormatsFWLite.so");
TFile f("....root");
.x print_data.C++
```

Old method using TFile directly

1. Start the autoloader
2. create the TFile
3. get the 'Events' TTree
4. call `GetEntry()` on the 'Events' TTree. This seems to help make sure the `SetAddress` calls will work
5. create instances of the classes for the ROOT branches you want to read
6. call `GetBranch` for each branch of the 'Events' Tree you want to read
7. for each branch, call `SetAddress` with the address of the class instances as an argument
8. when looping over the events
    1. call the `GetEntry` method of all the branches using the event index as the argument
    2. call the `GetEntry` method of the 'Events' TTree passing it the event index as the first argument and 0 as the second argument. A second argument of 0 tells ROOT not to read all the branches of the TTree.

An example macro template is shown below

```
{
   gSystem->Load("libFWCoreFWLite.so");
   FWLiteEnabler::enable();     // AutoLibraryLoader::enable() in CMSSW_7_4_X and prior releases

   TFile file("....root");

   TTree *events = (TTree*)file.Get("Events");
   //Needed for SetAddress to work right
   events->GetEntry();

   //set the buffers for the branches
   std::vector<...> data;
   TBranch* dataB = events->GetBranch("..._..._..._....obj")
   dataB->SetAddress(&data);

   //loop over the events
   for( unsigned int index = 0;
          index < events->GetEntries();
          ++index) {
      //Need to reset address just in case this same branch is read by an edm::Ref
      dataB->SetAddress(&data);
      dataB->GetEntry(index);
      events->GetEntry(index,0);

      //now can access data
      std::cout <<" size "<<data.size()<<std::endl;
      ...
   }
}
```

Using a List of Files

Using fwlite::ChainEvent

The fwlite::ChainEvent allows you to use the same information you use when accessing data in cmsRun using edm::Event.

1. Start the autoloader
2. load in the helper library `gSystem->Load("libDataFormatsFWLite")`
3. create a `std::vector<std::string>` to hold the list of file names
4. include "DataFormats/FWLite/interface/Handle.h" to get the `fwlite::Handle`
5. `push_back` each file name into the `std::vector<std::string>`
6. create an `fwlite::ChainEvent` by passing to the constructor the `vector`
7. create a for loop
    1. start loop by calling `toBegin()` on the `fwlite::ChainEvent`
    2. for each iteration of the loop call `atEnd` on the `fwlite::ChainEvent`
    3. at the end of each iteration, increment the `fwlite::Event` by using the operator++ method
8. when looping over the events
    1. create an `fwlite::Handle< ... >` where the template argument is the C++ class you want to get from the event
    2. call the `getByLabel` method of the `fwlite::Handle` passing it the event and the strings used to denote the object

An example macro template is shown below

```
{
   gSystem->Load("libFWCoreFWLite.so");
   FWLiteEnabler::enable();     // AutoLibraryLoader::enable() in CMSSW_7_4_X and prior releases
   gSystem->Load("libDataFormatsFWLite.so");

   #include "DataFormats/FWLite/interface/Handle.h"
   vector<string> fileNames;
   fileNames.push_back("....root");

   fwlite::ChainEvent ev(fileNames);

   for( ev.toBegin();
          ! ev.atEnd();
          ++ev) {
       fwlite::Handle<std::vector<...> > objs;
       objs.getByLabel(ev,"....");
      //now can access data
      std::cout <<" size "<<objs.ptr()->size()<<std::endl;
      ...
   }
}
```

Compling the Macro with ACliC

The compilation is identical to what is done for the case of a single file above, except in the macro you replace the use of `fwlite::Event` with `fwlite::ChainEvent`. NOTE: You must still pick one of the files to be used in the `fwlite::ChainEvent` and open it with a `TFile` on the ROOT command line in order to force the proper dictionaries open.

Old method using TChain directly

In the 1_2_0_pre series it became possible to use FWLite with a TChain

1. Start the autoloader
2. create the TChain with the Tree name 'Events'

3. create instances of the classes for the ROOT branches you want to read
4. create pointers to TBranch objects for each branch you want to read
5. call `SetBranchAddress` for each branch of the 'Events' TChain you want to read
6. when looping over the events
    1. for each branch, call `SetAddress` with the address of the class instances as an argument. This is needed since each time the TChain changes files the TBranch is changed
    2. call the `GetEntry` method of all the branches using the event index as the argument
    3. call the `GetEntry` method of the 'Events' TChain passing it the event index as the first argument and 0 as the second argument. A second argument of 0 tells ROOT not to read all the branches of the TChain.

An example macro template is shown below

```
{
   gSystem->Load("libFWCoreFWLite.so");
   FWLiteEnabler::enable();     // AutoLibraryLoader::enable() in CMSSW_7_4_X and prior releases

   TChain events("Events")

   //set the buffers for the branches
   std::vector<...> data;
   TBranch* dataB;
   events->SetBranchAddress("..._..._..._....obj",&data,&dataB)

   //loop over the events
   for( unsigned int index = 0;
          index < events->GetEntries();
          ++index) {
      //need to call SetAddress since TBranch's change for each file read
      dataB->SetAddress(&data);
      dataB->GetEntry(index);
      events->GetEntry(index,0);

      //now can access data
      std::cout <<" size "<<data.size()<<std::endl;
      ...
   }
}
```

Using ROOT to compile a macro

The first step that ROOT takes when compiling a macro is to run the *CINT* interpreter over the macro in order to determine what class or function 'dictionaries' it must create. After that step, the regular C++ compiler is used to build the code. Unfortunately, *CINT* is incapable of properly parsing many of our header files. However, it turns out the headers are not needed by *CINT* but only by the compiler, therefore adding

```
#if !defined(__CINT__) && !defined(__MAKECINT__)
...
#endif
```

around the header files avoids the problem with *CINT*.

However, the compiler still needs to know where to find our header files. FWLite pre-configures ROOT to find CMS headers from the environment variables `CMSSW_BASE` and `CMSSW_RELEASE_BASE`. FWLite pre-configures ROOT to find standard externals header files (e.g. boost and CLHEP).

TFWLiteSelector

A TSelector is ROOT's way of writing the equivalent of an EDM's EDAnalyzer, i.e. a C++ class which gets data from the file and then processes it. TSelectors are also the only way to use a PROOF farm (i.e. ROOTs way of doing parallel processing). TFWLiteSelector inherits from TSelector but allows you to access data using the standard edm::Event. This means you can use the exact same code you use in an EDAnalyzer from within ROOT.

The easiest way to start to write a TFWLiteSelector is to use the 'mktsel' command from the shell. From within a subsystem directory of your SCRAM project area, do

```
mktsel <name>
```

Where is the name you want to give to your TSelector class (it will also be the name of the SCRAM package which `mktsel` will create). Once created, you can edit the generated code and then compile it using SCRAM (I was unable to make ROOT's ACliC compiler to work with the class). Then in ROOT,

1. load the library containing your TSelector using `gSystem->Load("<your library name>")`
2. build a TChain
3. call the `Process` method of the TChain, passing either a pointer to an instance of your TSelector or a string containing the name of your TSelector

Examples of working TFWLiteSelectors can be found in CMSSW at FWCore/TFWLiteSelectorTest/src with the ROOT macros. However, the files generated using `mktsel` are meant to contain enough documentation to get you started properly.

Known bugs

CINT (ROOT command-line or macro) does not know symbol if it includes a `std::`

If a templated class contains a template argument which has the namespace `std::` then CINT will not be able to find the type. E.g.,

```
root [2] edm::Wrapper<std::vector<reco::Vertex> > t
Error: Symbol Wrapper<std::vector<reco::Vertex> >t is not defined in current scope  (tmpfile):1:
*** Interpreter error recovered ***
```

This can be solved by removing the `std::` from the type

```
root [3] edm::Wrapper<vector<reco::Vertex> > t
```

NOTE: This is a known problem with CINT and the ROOT team has informed us that they will be fixing this in a future version of ROOT as part of a major internal change.

CINT (ROOT command-line or macro) does not know `edm::Wrapper<vector<...> >` where `...` is a simple type

The way we generate ROOT dictionaries (via Reflex) for *std::vector* of simple types does not agree with how CINT generates dictionaries for the same type. This means if CINT sees

```
edm::Wrapper<vector<int> > wi;
```

it will not be able to find the dictionary and will generate an error like

```
Error: Symbol Wrapper<vector<int,allocator<int> > > is not defined  in current scope
```

It is possible to 'trick' CINT into finding the correct dictionary by executing the following command before trying to use the Wrapper type

```
gROOT->ProcessLine("namespace edm {typedef  edm::Wrapper<vector<int>>  Wrapper<vector<int,
```

edm::Ref's fail if user manual reads the same TBranch as the edm::Ref

Prior to release CMSSW_1_5_X if a user read from the same TBranch as an edm::Ref, the data gotten by the edm::Ref would be 'stale'. A fix was backported to the CMSSW_1_3_X series and can be obtained by checking out from CVS FWCore/FWLite with the tag V00-09-08-01.

Review Status

Reviewer/Editor and Date (copy from screen)	Comments
ChrisDJones - 15 Aug 2006	page author
BenediktHegner - 31 Jan 2007	page last content author
JennyWilliams - 01 Feb 2007	editing to include in SWGuide
WilliamTanenbaum - 2015-06-16	AutoLibraryLoader renamed to FWLiteEnabler (7_5_X)

Responsible: ChrisDJones
Last reviewed by: Reviewer

This topic: CMSPublic > SWGuideEDMWithRoot
Topic revision: r27 - 2015-06-16 - WilliamTanenbaum